Notes on MTK

Madhusudhan Pandey, PhD

November 20, 2025

Abstract

This lecture provides an overview of the standard repository structure used in modern scientific Julia packages, using ModelingToolkit.jl as a representative example. The session introduces graduate and PhD students to the essential folders, configuration files, and automation tools that support high-quality research software development. Key concepts include documentation workflows, benchmarking, modular extensions, testing infrastructures, and continuous integration pipelines. By understanding the purpose and interplay of components such as `src`, `docs`, `test`, and the `.github` automation ecosystem, students gain the foundational skills required to design, maintain, and publish professional scientific software. The lecture equips learners to build reproducible, scalable, and maintainable research codebases aligned with best practices in computational science and engineering.

This lecture notes has several modules.

This is Lecture Module 2.

The File `src/ModelingToolkit.jl`: Core Engine of the Package

The file `src/ModelingToolkit.jl` serves as the central engine of the `ModelingToolkit.jl` package. For graduate and PhD students studying scientific software architectures, this file is a canonical example of how a large Julia package organises, loads, and refine its core computational capabilities.

Although the full functionality of the package spans many subdirectories and files, the main module file acts as the root controller:

- declaring the module.
- importing julia files
- exporting public APIs
- loading subsidiary components
- symbolic compiler pipeline

Lecture Duration

Total Time: 30 minutes **Breakdown:**

- 5 min – Conceptual overview of the main module `ModelingToolkit`.
- 15 min – Submodules, imports, exports (API) and construction of the main Julia package module.

0.1 Learning Goals

- Understand the purpose and role of the main module file in a Julia package.
- Analyse how `ModelingToolkit.jl` integrates its core components from the `src/` directory.
- Recognise patterns for structuring scientific software and symbolic compilers.
- Apply these patterns when designing packages for modelling, simulation, or domain-specific computational workflows.

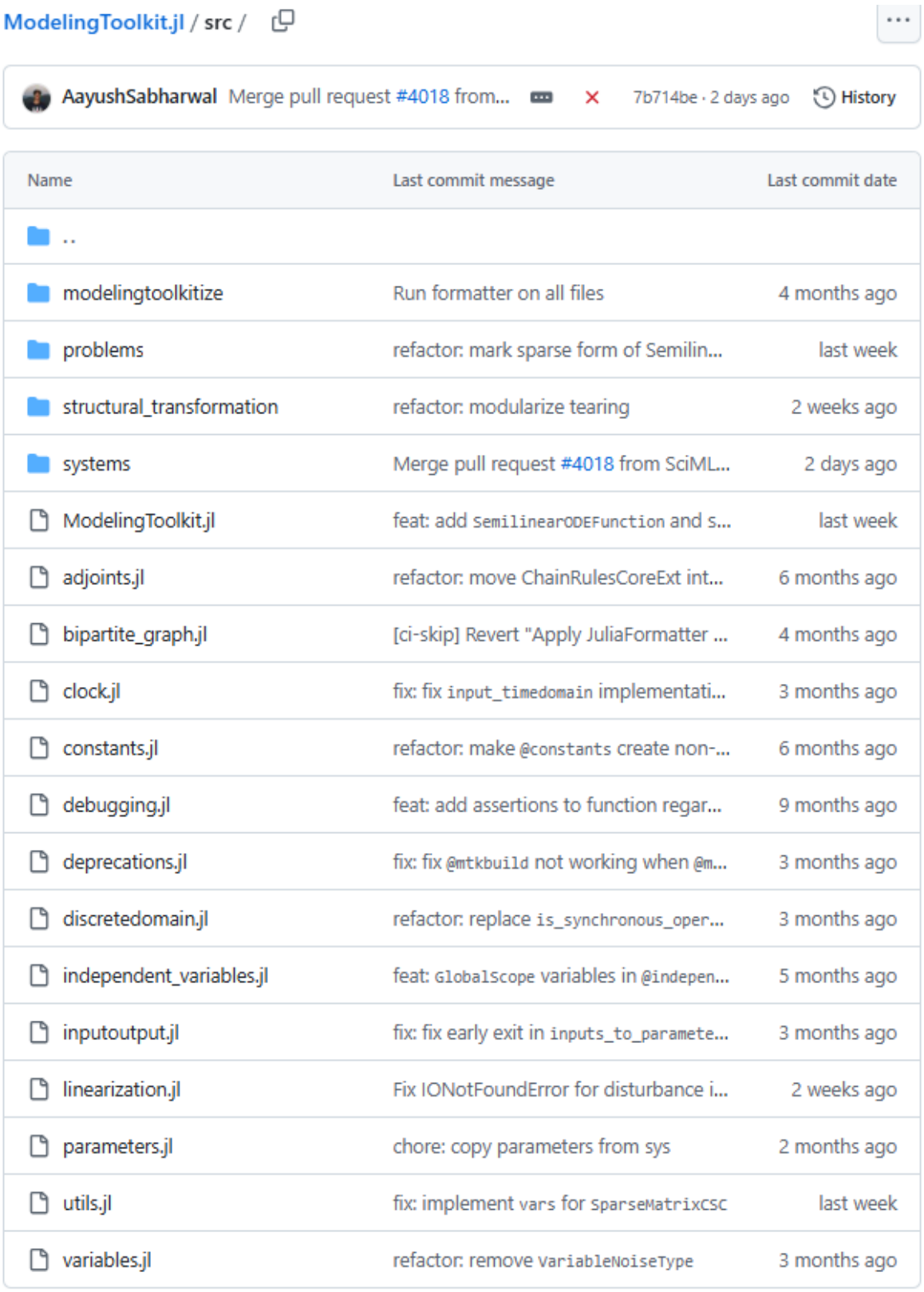

Figure 1: `ModelingToolkit.jl` main module with other subsidiary folders and files. All these files and folders are included inside the main module *ModelingToolkit*

0.2 Overview of the Main Module File

The file `ModelingToolkit.jl` defines the root module:

- Declaring the `ModelingToolkit` module.
- Importing external Julia libraries such as `Symbolics`, `LinearAlgebra`, and `SparseArrays`.
- Exporting core user-facing functionality such as `@variables`, `@parameters`, `ODESystem`, `SDESystem`, and `mtkcompile`.
- Loading internal subsystems via `include(...)` statements.

These responsibilities make the file the "entry point" of the symbolic modelling and compilation engine.

0.3 How the Engine Is Constructed

1. Module Declaration

The file begins with:

```
module ModelingToolkit
```

This establishes the namespace in which all symbols, types, and functions will reside. Everything included afterward becomes part of the module.

2. Imports and Using Statements

Standard Julia libraries and SciML tools are loaded here. These provide foundational functionality such as:

- symbolic expression manipulation,
- sparse matrix operations,
- differential equation infrastructure,
- graph representations of equation-variable systems.

3. Export Definitions

The exports define the public API of the package. Only exported symbols are intended for end-users, while others remain internal. This creates a clean separation between the modelling interface and the compiler internals.

4. Inclusion of Submodules

The heart of the file consists of numerous `include` statements such as:

```
include("variables.jl")
include("parameters.jl")
include("systems/odesystem.jl")
include("structural_transformation/tearing.jl")
include("linearization.jl")
include("utils.jl")
```

Through these statements, the module assembles components responsible for:

- defining symbolic variables and parameters,
- creating ODE, DAE, SDE, and hybrid systems,
- performing structural transformations such as tearing,

- building bipartite graphs for equation-variable analysis,
- executing linearization and compilation passes,
- handling model I/O and debugging.

This design mirrors the structure of domain-specific languages and compilers.

5. Internal Submodules

Some subsystems such as structural transformations, discretisation logic, or I/O interfaces are wrapped inside submodules declared inside the main file. This allows the package to logically group related code while maintaining a single overarching namespace.

6. Initialization

Many Julia packages define an `__init__()` function at the end of the main module. This function performs tasks that must occur after precompilation, such as caching symbolic rules or registering compiler passes.

0.4 Why This File Is the "Engine Room"

The main module file functions as the orchestration layer of the package. It:

- links together types, systems, and symbolic structures from throughout the `src/` directory;
- binds symbolic equations to their corresponding compiler passes;
- organises the modelling pipeline from variable creation to problem generation;
- presents a unified user-facing API, even though functionality is distributed across many files.

Without this coordinating file, the complexity of the package would be fragmented across dozens of source files, making it impossible for users or developers to interact with the toolkit coherently.

0.5 Summary

The file `src/ModelingToolkit.jl` is the central coordinating element of the entire package. It declares the module, exposes the API, includes component files, and defines the symbolic modelling environment. As such, it represents the "engine" that binds together symbolic systems, differential-algebraic formulations, compiler passes, structural transformations, linearisation mechanisms, and utilities. Understanding this file is essential for students learning how large-scale Julia packages are architected and how symbolic modelling software is engineered.